RISCSmith: Finding RISC-V CPU Bugs via Rich Instruction Construction and On-the-fly Differential Analysis

Xudong Zhang[†], Yuanliang Chen[†✉], Zehong Yu[†], Zhen Yan[†], Fuchen Ma[†], Dalong Shi[‡], Yu Jiang[†✉]
[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[‡] Aviation Industry Corporation of China Ltd., Beijing, China

Abstract

Processors are the foundation of all computing systems, yet complex RTL and microarchitectural front ends make logic defects difficult to eliminate and extremely costly to fix after tape-out. Although RISC-V fuzzing has revealed many bugs in practice, existing approaches remain limited: they rely on manually maintained instruction models and generate insufficiently rich CPU test programs. Moreover, they lack precise register and memory monitoring, and either perform high-overhead per-instruction differential checks or coarse end-of-program comparisons – both making bug analysis and localization inefficient.

In this work, we propose RISCSmith (30K+ Rust LoC), a fuzzing framework aimed at detecting bugs in RISC-V CPUs. First, RISCSmith automatically builds rich instruction models by parsing the RISC-V UnifiedDB to extract structured instruction metadata, resolve inconsistencies, and generate strongly typed models capturing operand roles and runtime semantics. Second, RISCSmith instruments RISC-V implementations with lightweight logging to collect per-instruction register, memory, and exception data, performing on-the-fly differential analysis to pinpoint the first divergence, classify its cause, and minimize the reproducing test case. We implemented and evaluated RISCSmith on six widely used RISC-V CPUs. In total, it uncovered 18 previously unknown bugs. Compared to state-of-the-art CPU fuzzers like Cascade and RISCV-DV, RISCSmith detects 3.5x and 2.6x more bugs and covers 37% and 61% more branches, respectively.

ACM Reference Format:
Xudong Zhang[†], Yuanliang Chen[†✉], Zehong Yu[†], Zhen Yan[†], Fuchen Ma[†], Dalong Shi[‡], Yu Jiang[†✉]. 2026. RISCSmith: Finding RISC-V CPU Bugs via Rich Instruction Construction and On-the-fly Differential Analysis. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803916

1 Introduction

Processors form the foundation of modern computing systems. The growing complexity of their design–such as multi-stage pipelines and speculative execution [6, 8, 40]–has led to numerous security vulnerabilities [19, 21, 25]. Detecting such bugs before fabrication is critical to prevent severe consequences, including substantial financial losses [1, 18]. However, exhaustive pre-silicon verification (i.e., testing all possible scenarios) remains infeasible due to the vast state space of modern processors [5].

Yuanliang Chen and Yu Jiang are the corresponding authors.

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803916

With the increasing adoption of open-source RISC-V CPUs and the high cost of formal verification, hardware fuzzing has gained significant traction [5, 17, 20, 31, 34, 38, 39]. CPU fuzzers typically generate random instruction sequences or mutate existing programs to explore new processor behaviors, and then compare execution against a reference model such as Spike [30] to detect state differences and reveal design defects. Existing fuzzers have uncovered many bugs in real processor implementations, including Rocket [9], BOOM [3], and CVA6 [11], improving the security and reliability of the RISC-V hardware ecosystem.

However, current tools face the following limitations. (1) **Incomplete Instruction Modelling:** Given RISC-V continues to evolve rapidly, hand-maintained instruction models struggle to completely cover full instruction sets, leaving substantial functionality unexploited. Generated programs often lack proper register or memory initialization (e.g., memory instructions using base registers that were never assigned valid addresses), and instructions rarely interact on shared data, making it difficult to form meaningful dependencies. As a result, the overall quality of the test cases remains limited. (2) **Insufficient Bug Detection:** For bug detection, existing approaches either halt after every instruction to compare architectural state (e.g., register values), which introduces significant runtime overhead, or compare only at program completion, which misses the first point of divergence and the detailed mismatch. In addition, while these methods typically perform precise comparison of register differences, memory checking is often coarse-grained and can overlook subtle errors.

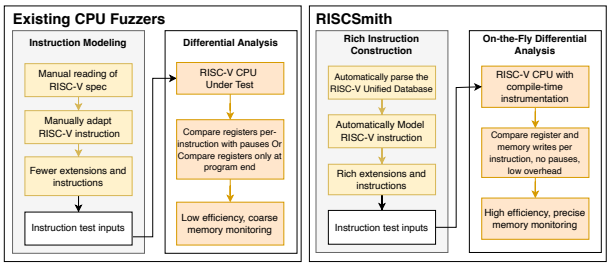

Figure 1: Comparison between RISCSmith and existing CPU fuzzers. RISCSmith automatically models rich RISC-V instructions and efficiently analyzes fine-grained memory and register differences with low overhead.

In this work, we propose RISCSmith, a framework for detecting RISC-V CPU bugs through rich instruction construction and on-the-fly differential analysis. Our key insight is to leverage the RISC-V UnifiedDB to generate rich instruction sequences based on authoritative and up-to-date instruction specification, and employ low-instruction logging instrumentation to enable efficient differential analysis. (1) To enable rich instruction modelling, RISCSmith first parses the RISC-V UnifiedDB, and authoritative and up-to-date specification, to extract

concise and structured instruction metadata, including instruction encoding, operands and access semantics. RISCSmith then reconciles naming difference, resolves inconsistencies, and constructs strongly typed instruction models that capture operand roles, type constraints, and runtime semantics. (2) To enable high-efficiency bug analysis, RISCSmith instruments RISC-V implementations with lightweight logging hooks. RISCSmith captures memory updates, register writes, and exceptions to reconstruct instruction execution context, and performs on-the-fly comparison between the target implementation and the reference model to localize the first divergence. Finally, RISCSmith determines dependency relationships leading to the mismatch, classifies the divergence type, and automatically minimizes the instruction sequence into a reproducible test case.

We conducted experiments on six real-world RISC-V CPUs, i.e. PicoRV32 [14], CVA6 [11], VexRiscv [13], Rocket [9], Xiangshan [12], and Kronos [35]. RISCSmith has discovered 18 new bugs. Compared with other SOTA CPU fuzzers such as Cascade [34], and RISCV-DV [31], RISCSmith detects 3.5x and 2.6x more bugs, covers 37% and 61% more branches. In summary, we make three contributions:

- We propose a rich RISC-V instruction model automatically constructed from the UnifiedDB.
- We design an on-the-fly differential analysis with low-overhead instrumentation to monitor differences precisely.
- We implement and evaluate RISCSmith on six widely used RISC-V CPUs. We will open-source it[1] for practical usage. Currently, it has already detected 18 new bugs.

2 Background And Motivation

RISC-V is an open, modular ISA whose base and rich standard extensions (M/A/F/D/C/V/H and many Z*/S* series) support deployments from embedded devices to high-performance computing, and whose open ecosystem has produced diverse implementations such as Spike, Rocket, CVA6, and XiangShan. To keep these rapidly evolving implementations aligned with the specification, the RISC-V Unified Database (UnifiedDB) provides an official, unified, and trustworthy data source that precisely describes, using structured YAML and JSON Schema, the encoding, operand constraints, and IDL semantics of all standard extensions, instructions, and CSRs. Its modular versioning and support for custom extensions ensure traceability of spec evolution and extensibility, while giving data-driven modeling and verification tools a stable foundation that automatically inherits the latest definitions as the RISC-V instruction specification evolves.

In large RISC-V processor projects, pipeline, memory, and permission control logic are tightly coupled. If pre-silicon verification misses such issues, architecture-level divergences discovered after tape-out may require costly ECOs; hence, strong verification confidence must be achieved early. A memory-permission defect we found in CVA6 illustrates this risk. Figure 2 compares three scenarios that differ only in whether PMP CSR probing (Prefix) and the hypervisor load `hlvx.hu` (Suffix) surround the same user-mode store (Core). Each column shows the Prefix CSR reads, the Core store, and the Suffix load, with colors marking trapping and non-trapping instructions. In Scenario A (bug), CSR reads such as `mseccfg`, `pmpcfg`, and `pmpaddr` precede a user-mode store followed by `hlvx.hu`; CVA6 incorrectly traps the store as `LD_ACCESS_FAULT` even though the address is valid in user space, while Spike and Rocket both execute it correctly. In Scenarios B and C, where either the Prefix or Suffix is removed, the store succeeds. Only by reproducing the dependency between surrounding instructions can differential comparison expose this deviation.

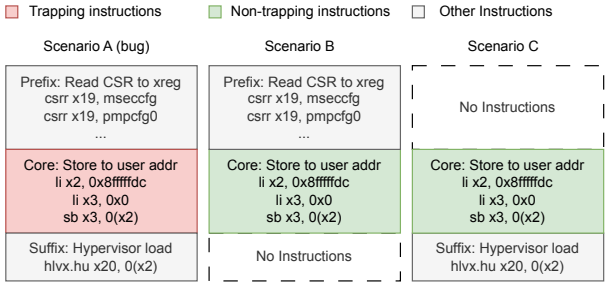

Figure 2: CVA6 memory-permission bug: the store triggers `LD_ACCESS_FAULT` only in Scenario A, where it is enclosed by PMP CSR (Prefix) and the hypervisor load `hlvx.hu` (Suffix).

This bug exposes two key weaknesses in existing RISC-V testing methods and the improvements needed. (1) Manual instruction modeling cannot keep up with expanding extensions – unsupported operations like `hlvx.hu` often lack proper register or memory setup, leaving instructions to access inconsistent addresses. To overcome this, automated modeling is needed to cover rich instructions, generate constrained programs, and prepare valid contexts for multi-instruction interaction. (2) Coarse differential comparison is either precise but slow with per-instruction pauses or fast but incomplete when checking only end states, missing early divergences such as misclassified `sb` traps. This calls for high-speed execution with fine-grained logging that captures per-instruction register, memory, and exception data, precisely localizing divergences with low overhead.

3 Design

Design goal: A practical RISC-V CPU fuzzing framework should have the following properties.

- *General:* RISCSmith is designed to detect bugs in a wide range of practical RISC-V CPUs and supports mainstream RISC-V architectures such as RV32IMC, RV64IMAC, RV64GC, and RV64IMAFDC. The tool can be rapidly deployed to test different RISC-V CPU architectures with only minor configuration adjustments.
- *Efficient:* RISCSmith is able to frequently exercise the CPUs and effectively detect bugs in real-world RISC-V CPUs in 24 hours.
- *Accurate:* RISCSmith is designed to have satisfying precision and recall to avoid reporting false positives.

3.1 RISCSmith Workflow

Figure 3 illustrates the workflow of RISCSmith, consisting of two main stages. **Instruction Modelling stage.** (1) RISCSmith first parses the RISC-V UnifiedDB to extract each instruction's encoding, operands, immediates, and access semantics, producing structured instruction data. (2) It then reconciles naming differences, resolves inconsistencies, and builds strongly typed instruction models that encode operand roles, type constraints, and runtime semantics. (3) These models are integrated into a unified *rich RISC-V instruction model*, which serves as the foundation for the **testcase generator**. The generator automatically constructs programs that initialize register and memory contexts, embed inter-instruction dependencies, and insert exception carry-on templates. **Differential Testing stage.** (4) Source instrumentation. RISCSmith instruments the RISC-V implementation source by inserting lightweight logging hooks into the RTL. The instrumented code

[1]RISCSmith at: https://anonymous.4open.science/r/RISCSmith-EB85

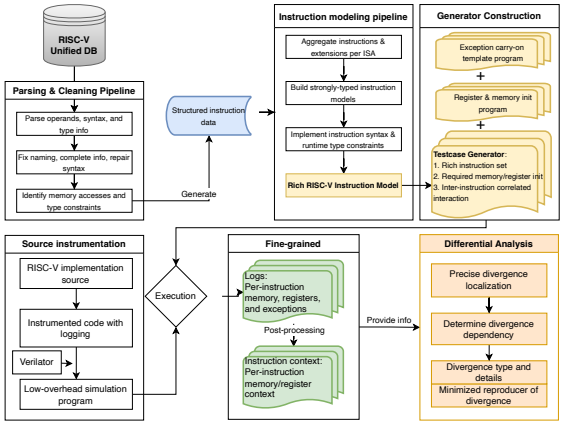

Figure 3: The workflow of RISCSmith. It includes two main processes: (1)Rich RISC-V Instruction Automatic Construction for generating high-quality instruction test inputs; (2) On-the-fly Differential Analysis for detecting bugs in CPUs.

is compiled and simulated using Verilator to produce a low-overhead execution environment capable of recording per-instruction events. (5) Fine-grained logging. During execution, RISCSmith captures detailed logs of per-instruction memory updates, register writes, and exceptions. The collected data is then post-processed to reconstruct each instruction's execution context, including its memory and register state, forming a precise behavioral trace for later comparison. (6) Differential analysis. RISCSmith performs on-the-fly comparison between the target implementation and the reference model to localize the first divergence. It determines dependency relationships leading to the mismatch, classifies the divergence type, and automatically minimizes the instruction sequence into a reproducible test case. This process enables precise and efficient identification of semantic inconsistencies across different RISC-V CPU implementations.

3.2 Rich RISC-V Instruction Construction

RISCSmith's modeling stack is fully data driven. Every UnifiedDB entry is ingested to capture the owning extension, supported widths, encoding fields, register aliases, immediate magnitudes, parity requirements, and forbidden constants. The importer reconciles stylistic differences across source documents, normalizes naming and letter case, fills in missing text when schema hints allow, and records provenance whenever conflicts arise so downstream stages always consume a coherent, auditable dataset. For base-plus-offset instructions, the assembler grammar is further dissected to flag fixed registers, selectable registers, and pure offset fields so all consumers see the same trustworthy "instruction truth."

With this canonical view in place, the code generator evaluates whether each instruction is shared between RV32 and RV64, then emits layered type hierarchies that cover shared views as well as architecture-specific splits. Every instruction receives its own data structure; constrained registers or immediates are rewrapped into bounded types that carry parity rules and forbidden sets, and display logic, legality checks, randomization hooks, and configuration wiring are composed alongside them. A centralized configuration surface exposes register windows, memory ranges, immediate domains, and curated register lists so users can adjust generation policies without touching code—for instance limiting integer destinations to x8–x15 or pinning memory accesses inside a known window. Integer, floating-point, CSR, shadow-stack, and compressed-instruction registers all follow the same validation rules, and configurations can be shared across projects as reusable templates that dramatically shorten tuning time.

During program synthesis the randomizer analyzes dependencies per instruction. Memory operations automatically receive "prepare base and offset" snippets so computed addresses always fall inside managed regions rather than accidentally reusing stale register values. CSR, atomic, and arithmetic/logic instructions operate within the same narrow register and memory windows, forcing sequences to reuse state; this deliberate crowding causes instructions to read, modify, and rewrite the same locations repeatedly, which stresses coherence, protection, and exception handling far more effectively than blindly increasing sequence length.

To ensure the generated programs remain practical, RISCSmith continuously assembles batches of random instructions that span multiple extensions with the standard GNU RISC-V toolchain. Any failing samples, build logs, and minimized reproducers are archived so the modeling team can react immediately. Custom extensions reuse the identical pipeline: providing UnifiedDB-style metadata is enough to regenerate models, randomization policies, and runnable tests at the same quality level as official extensions—no template rewrites or ad-hoc scripts required.

3.3 On-the-fly Differential Analysis

RISCSmith conducts differential testing through a three-stage pipeline that integrates low-overhead instrumentation, fine-grained execution monitoring, and automated bug analysis. This design ensures accurate and reproducible divergence detection while keeping runtime overhead minimal, enabling continuous high-throughput testing across RISC-V CPU implementations.

Lightweight Source Instrumentation. RISCSmith first instruments the RISC-V implementation source with minimal logging hooks that record per-instruction register and memory writes as well as exception events. The instrumented RTL is compiled through Verilator into a low-overhead simulation executable, ensuring that instruction-level traces can be collected efficiently without disrupting the CPU's original execution timing or semantics. It establishes the foundation for accurate post-execution context reconstruction.

Fine-grained Difference Detection. During simulation, RISCSmith captures detailed per-instruction logs, including memory updates, register writes, and exceptions. After execution, these logs are normalized into unified instruction contexts representing the "before–instruction–after" state of every executed operation. The system then compares these contexts across multiple implementations or against a golden reference model to precisely detect divergences in exception behavior, register state, or memory contents. This fine-grained comparison enables early identification of semantic mismatches with minimal false positives.

On-the-fly Bug Analysis. When RISCSmith detects a behavioral difference, whether in register state or memory content, it automatically logs all executed RISC-V instructions from the target CPU, timestamping and organizing them into a structured reproduction log. Using this log, RISCSmith attempts to deterministically reproduce the divergence and applies a binary search–based minimization algorithm to isolate the shortest instruction sequence that consistently triggers

Table 1: 18 new bugs detected by the tools within 24 hours. RISCSmith found all 18 bugs. In comparison, Cascade found 4 bugs, and RISCV-DV found 5 bugs, respectively. Identifiers are reported using public issue numbers.

#	RISC-V CPUs	Root Cause Analysis	Location	Identifier
1	PicoRV32	Misaligned load trap still commits a stale register writeback.	Load misalignment trap	PICO-278
2	PicoRV32	Misaligned store trap still drives AXI write strobes and corrupts memory.	AXI-Lite write strobes	PICO-279
3	PicoRV32	A byte load can read a stale byte from an older word, diverging from Spike.	Byte-load data mux	PICO-283
4	CVA6	AMO instructions never log the memory write in RVFI, only the register result.	RVFI AMO trace logic	CVA6-3130
5	CVA6	Div/sqrt ignores NaN-boxing, so unboxed NaNs are treated as normal numbers.	CVFPU div/sqrt NaN-box path	CVA6-3123
6	CVA6	RV32 div/sqrt truncates and re-boxes results, making `fsqrt.s` return NaN-boxed zero.	RV32 single-precision sqrt datapath	CVA6-2449
7	CVA6	FPU-to-integer ops write an integer register but the trace tags a FP register.	RVFI int/FP writeback tagging	CVA6-3122
8	CVA6	After PMP probing, `sb` followed by `hlvx.hu` is reported as a load fault instead of a store fault.	PMP cache / LSU exception type	CVA6-3126
9	CVA6	Hypervisor misaligned loads (e.g., `hlv.d`) encode the store/AMO misaligned cause instead of load misaligned.	Hypervisor load cause encoder	CVA6-3146
10	CVA6	Hypervisor misaligned stores (e.g., `hsv.d`) use a generic fault code instead of store/AMO misaligned.	Hypervisor store cause encoder	CVA6-3147
11	CVA6	With Zfa+FP16ALT, `rm[2]` is shared, so `fleq.*` can treat NaNs as normal FP16ALT values and return 1.	FPU compare unit rm[2] mux	CVA6-3145
12	VexRiscv	FSGNJX.D uses a NaN-boxed rs2 with a cleared sign bit, so the XOR sign is wrong.	FPU SGNJ sign/NaN-box path	VEX-464
13	VexRiscv	Writes to read-only CSRs `cycle`/`instret` retire without illegal-instruction.	CSR read-only counter check	VEX-465
14	Rocket	RV32D `fld` uses 32-bit `data_word_bypass`, leaving the upper 32 bits of FP regs stale.	FPU load data source	ROCKET-3773
15	XiangShan	For `hlvx.hu`, the TLB/exception logic prefers load guest-page fault (0x13) over load access fault (0x5).	MMU `hlvx` permission / cause	XS-5137
16	XiangShan	AMOs on misaligned addresses raise store address misaligned (0x6) instead of store access fault (0x7).	AMO exception vector / priorities	XS-4667
17	XiangShan	`hvictl` and `time` are treated as writable CSRs instead of trapping illegal writes.	NewCSR hypervisor/time CSR checks	XS-5248
18	Kronos	Counter CSRs `cycle`/`time`/`instret` accept writes and do not raise illegal-instruction.	Counter CSR write/illegal check	KRONOS-20

the anomaly. If the minimized sequence reproduces the issue reliably, RISCSmith classifies it as a confirmed bug. Leveraging the fine-grained source instrumentation introduced earlier, RISCSmith then maps each instruction input back to its corresponding RTL execution trace, rapidly identifying the faulty logic and performing automated bug deduplication across different runs and CPU architectures to ensure precise, scalable fault analysis.

4 Implementation and Evaluation

We implement an automated verification pipeline, named RISCSmith (about 30K+ lines of Rust code), which spans from instruction modeling to differential testing. RISCSmith uses the official RISC-V UnifiedDB as its primary data source. Through automated parsing and code generation, it constructs constrained and strongly typed instruction models that serve as the basis for a random program generator. The generator automatically prepares register and memory contexts, models inter-instruction data dependencies, and includes an exception propagation mechanism. To balance execution efficiency and analytical precision, we integrate lightweight logging into the RTL and, during Verilator simulation, record per-instruction register updates, memory writes, and exceptions with minimal overhead. The framework conducts differential log analysis to locate the first behavioral divergence, identify register and memory mismatches, and automatically generate minimized reproducing test cases, achieving a practical trade-off between large-scale validation and fine-grained debugging.

In this section, we evaluate RISCSmith to answer the following three research questions:

- **RQ1**: Is RISCSmith effective in detecting vulnerabilities of real-world RISC-V CPUs?
- **RQ2**: Can RISCSmith cover more code of RISC-V CPUs compared with state-of-the-art methods?
- **RQ3**: Does the on-the-fly differential analyzer effectively improve testing performance?

4.1 Experiment setup

Subject: We evaluated RISCSmith on six real-world RISC-V CPUs with diverse architectural configurations. Table 2 summarizes the detailed information of these evaluation targets. The selection of CPUs was based on two criteria: (1) They are widely deployed in practical environments, demonstrating RISCSmith's ability to uncover bugs that could affect a broad spectrum of users; and (2) they represent heterogeneous RISC-V architectures, including RV32IMC, RV64IMAC, RV32IM, RV64IMAFDC, RV64GC, and RV32IMC/RV32EC, showcasing RISCSmith's generality across different pipeline depths, ISA extensions, and implementations.

Table 2: Detailed information of target RISC-V CPUs

CPU Name	Architecture	Version	Vendor
PicoRV32	RV32IMC	Master, 87c89ac	YosysHQ
CVA6	RV64IMAC	Master, 89986df	OpenHW Group
VexRiscv	RV32IM	Master, b91c909	SpinalHDL
Rocket	RV64IMAFDC	Master, 8f1e33b	CHIPS Alliance
XiangShan	RV64GC	Master, 56fe218	OpenXiangShan
Kronos	RV32IMC/RV32EC	Master, b0c06eb	lowRISC

Compared Tools: We compared RISCSmith with two state-of-the-art RISC-V CPU fuzzers: Cascade [34] and RISCV-DV [31]. Cascade performs coverage-guided fuzzing on RTL models using Verilator feedback to mutate instruction sequences, while RISCV-DV employs constrained-random generation within the SystemVerilog UVM framework to produce coverage-driven instruction streams. These two tools were chosen as baselines because they represent the most established approaches to RISC-V CPU fuzzing, Cascade emphasizing feedback-directed RTL fuzzing and RISCV-DV serving as the industry-standard random test generator.

Metrics and Settings: We employed three metrics for our evaluation: the number of unique bugs detected, the code coverage of target CPUs, and the average speed-up to localize these bugs. These metrics are commonly used to measure the effectiveness of fuzzers by prior researches [4, 15, 34]. We ran each testing tool under the same environment setup on a computer equipped with an Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, 32 GiB of DDR4 memory, and running x86_64 Ubuntu Linux 20.04. All target RISC-V CPUs were tested using their default configuration parameters. All experiments were repeated 10 times under identical conditions, and the average results are used in this paper.

4.2 Bug Detection in Real-World RISC-V CPUs

We applied RISCSmith, Cascade, and RISCV-DV on all 6 target RISC-V CPUs for real-world bug detection in 24 hours. In total, RISCSmith discovered 18 previously unknown bugs across the six RISC-V CPUs, significantly outperforming other state-of-the-art fuzzers, none of which detected more than 5 bugs. Detailed information on these bugs is summarized in Table 1.

As shown in Table 1, RISCSmith discovered 18 previously unknown bugs across six representative RISC-V CPUs. Among these bugs, six

are related to exception or trap mis-encoding, which can cause the processor to report incorrect exception causes or fail to trigger traps properly, potentially leading to silent system errors or security policy violations. Four bugs are found in floating-point units (FPU), including incorrect handling of NaN boxing, faulty FP16/FP32 operand propagation, and sign-bit corruption in floating-point move operations, which can produce inconsistent computation results or break IEEE 754 compliance [27]. Three bugs involve CSR handling, where writes to read-only performance counters or system registers (e.g., `cycle`, `instret`, or `time`) were not trapped as illegal instructions–potentially allowing unauthorized modification of privileged states. In addition, two bugs were detected in load/store units, leading to misaligned access errors, stale register states, and corrupted data paths, while another two were found in AMO and memory logging units, revealing inconsistencies between hardware behavior and the reference model during atomic operations. Finally, one AXI-Lite interface defect caused incorrect write strobes on misaligned stores. Overall, these findings demonstrate that RISCSmith effectively exposes diverse categories of logic flaws spanning exception encoding, floating-point computation, CSR privilege enforcement, and memory access behavior across different RISC-V CPU architectures.

Comparison with existing Fuzzers: In our 24-hour experiments, Cascade and RISCV-DV identified only four and five bugs, respectively. However, the remaining 14 bugs were hidden in less frequently tested but critical CPU instruction-handling logic, which was only exposed by RISCSmith's rich instruction modeling, a capability absent in existing fuzzers. Unlike prior tools, RISCSmith automatically constructs rich RISC-V instruction models from the official UnifiedDB, enabling the generation of high-quality test programs that cover a wider range of CPU processing paths. Furthermore, through its on-the-fly, fine-grained differential analysis of memory and registers, RISCSmith successfully uncovered all 18 bugs, demonstrating its effectiveness and answering RQ1. Compared with other state-of-the-art CPU fuzzers, RISCSmith detected all the bugs found by them, while additionally revealing those hidden in complex instruction logic.

Table 3: Bugs found by each fuzzing tools. Others detect no more than 5 bugs, while RISCSmith detects all 18 bugs.

Tool	Bug Number	Bug IDs
RISCSmith	18	#1 – #18
Cascade	4	#2, #3, #9, #10
RISCV-DV	5	#2, #5, #9, #14, #16

4.2.1 *Case Study.*

We next present bug #11 from Table 1 as a case study illustrating how RISCSmith exposes semantic bugs in real RISC-V CPUs. As shown in Figure 4, a NaN-comparison defect in the CVA6 floating-point unit arises when the Zfa quiet-compare instructions (`fleq.`) are used together with the FP16ALT extension. The comparison front end reuses the rounding-mode bit `rm[2]` as both the Zfa "quiet compare" selector and the FP16ALT AH-format selector. For `fleq.`, this reuse can incorrectly force the comparison format to FP16ALT, causing a bit pattern that represents a quiet NaN in its true format (e.g., FP64) to be interpreted as a normal non-NaN value. As a result, `fleq.*` returns 1 instead of 0, where the RISC-V specification requires an unordered comparison.

This bug can be triggered by three conditions: (1) load a quiet-NaN bit pattern into a scalar floating-point register in a wide format

```
// Simplified excerpt: reuse of rm[2] in the compare front-end
always_comb begin
    // Scalar compare: operation encoded in rm[1:0]
--  cmp_mode  = rm[2:0];
++  cmp_mode  = {1'b0, rm[1:0]}; // compare mode ignores rm[2]
    // AH encoding: reuse rm[2] to select FP16ALT format
    if (!is_vector && check_ah) begin
--      if (rm[2]) begin
++      if (fp16alt_enable) begin
            dst_format = FP16ALT; // treat operands as FP16ALT
    ...
end
```

Figure 4: A NaN-comparison bug caused by reusing `rm[2]` for both Zfa quiet compare and FP16ALT format selection.

such as FP64; (2) enable both Zfa and FP16ALT and compare this register with itself using `fleq.*`; and (3) observe that the reference model (Spike) returns 0 (unordered) while CVA6, due to the `rm[2]` aliasing, returns 1 (equal), making this compare the first observable divergence. Such NaN misclassification can silently corrupt control flow in floating-point-intensive workloads. Figure 4 sketches the root cause in the compare front-end. In the simplified excerpt, the original implementation derives the compare mode directly from all three bits of `rm`, while the most-significant bit `rm[2]` is also reused to select FP16ALT via the AH-encoding logic:

In our experiments, this NaN-comparison defect was only detected by RISCSmith. Existing RISC-V CPU fuzzers rely on manually maintained instruction models and do not yet cover the Zfa `fleq.*` instructions, and they usually compare architectural state only at program end, so mismatched comparison results are easily overwritten. RISCSmith instead derives instruction models from the RISC-V UnifiedDB and therefore supports Zfa `fleq.d`/`fleq.s`/`fleq.h` together with FP16ALT configurations; its lightweight logging records, for every instruction, the operands, result, and architectural writes, and a post-processing pass performs instruction-by-instruction differential analysis against the reference model. As soon as a `fleq.*` result disagrees with Spike, RISCSmith pinpoints that compare as the first divergence and invokes automatic testcase minimization, reducing the input to a short sequence that constructs the NaN and executes a single `fleq.*`.

4.3 Effectiveness in Code Coverage

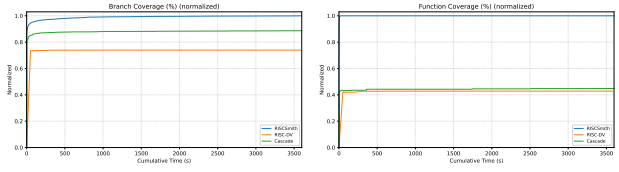

Figure 5: Code coverage over time. RISCSmith consistently covers more branches and functions compared to SOTA tools.

Since different tools implement distinct coverage instrumentation mechanisms, we used the evaluation by leveraging their shared capability to collect coverage data from the RISC-V golden model Spike [30]. To fairly evaluate the effectiveness of RISCSmith in terms of code coverage on RISC-V CPUs, we also adapted RISCSmith to Spike and used grcov [10] to collect coverage data over a 24-hour fuzzing period. The results, summarized in Figure 5, show that RISCSmith consistently outperforms all other state-of-the-art tools in both branch and function coverage. Compared with Cascade and RISCV-DV, RISCSmith achieves a 37%?1% improvement in branch coverage and a 123%?34%

improvement in function coverage. The main reason for this improvement is that RISCSmith supports a significantly broader and richer set of RISC-V instruction models. As shown in Figure 6, the number of instructions supported by RISCSmith is 2.3?.6x greater than that of Cascade and RISCV-DV. This instruction modeling enables the generation of more diverse and semantically complete test inputs, leading to higher code coverage across different CPU components. These results provide a clear and comprehensive answer to **RQ2**.

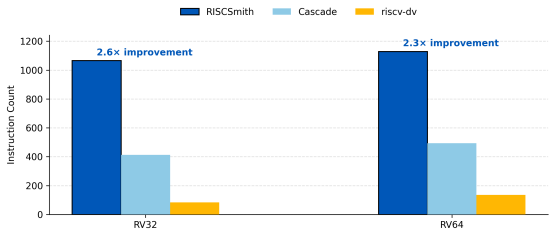

Figure 6: Number of RISC-V instructions supported by RISCSmith and other SOTA tools.

4.4 Effectiveness in On-the-fly Analysis

To evaluate the effectiveness of RISCSmith's on-the-fly differential analysis and assess the testing overhead introduced by its bug localization process, we measured the average time each tool required to perform bug localization after detecting a difference, as summarized in Table 4. Since RISCV-DV lacks an automated bug localization component and only reports raw differences that require manual analysis, it was excluded from the timing comparison. Compared with Cascade, which required an average of 5732.49 seconds for bug localization, RISCSmith achieved an average analysis time of 3065.91 seconds, representing a 46.5% improvement.

Table 4: Average Bug localization time used by fuzzers.

	Cascade	RISCV-DV	RISCSmith
Avg Bug Localization Time (ms)	5732.49	-	3065.91
Improvement	-	-	↑46.5%

Table 5: Overhead Introduced by tools' bug localization.

	Cascade	RISCV-DV	RISCSmith
Avg instruction execution (ms)	779.80	916.10	4080.19
Avg analysis process (ms)	442.64	4126.44	630.92
Overhead (%)	56.76%	450.44%	15.46%
Improvement	-	-	↑72.8%

We also measured each tool's average total instruction execution time during fuzzing, as well as the average time spent on differential analysis, as summarized in Table 5. RISCSmith's average instruction execution time was 4080.19 seconds, significantly higher than Cascade (779.8 s) and RISCV-DV (916.1 s), indicating that RISCSmith generates more complex instruction inputs capable of exercising deeper CPU processing logic. Despite this, the average time RISCSmith spent on differential analysis was much lower than that of RISCV-DV. The overhead of RISCSmith's on-the-fly differential analysis was only 15.46%, substantially lower than that of both Cascade and RISCV-DV. These results provide a clear and comprehensive answer to **RQ3**.

5 Discussion

Support for Private and Custom Extensions/Instructions. At present, RISCSmith relies on the RISC-V UnifiedDB to construct instruction models, and therefore only covers standard extensions and instructions. However, many processors introduce various private or custom extensions to support domain-specific acceleration (e.g., AI, DSP, cryptography) [7, 23, 28, 29, 32]. These extensions lack publicly available specification databases. In future work, we plan to extend the RISCSmith parser and generator, thereby enabling support for such private and custom extensions and instructions.

Extending the method to other domains. RISCSmith currently supports testing RISC-V CPUs, but its methodology can be extended to other ISAs (e.g., ARM, x86) and architectures (e.g., GPUs) [2, 22, 33]. However, differences in ISA semantics and processor execution models make direct cross-domain migration technically challenging. To address these issues, we plan to develop a more general instruction-modeling workflow and cross-architecture logging and differential-analysis tools to enable verification across heterogeneous architectures [16, 36, 37].

6 Related Work

CPU Fuzzing. As one of the most complex hardware components, CPUs have inspired many specialized fuzzing techniques. DIFUZZRTL [17] performs coverage-guided differential testing by aligning per-instruction states, while Cascade [34] generates long RISC-V programs and reduces overhead via non-termination checks. TheHuzz [20] and INSTILLER [39] apply hardware-aware coverage metrics, and MorFuzz [38] improves semantic relevance through runtime instruction morphing. However, these tools rely on manually built instruction models. In contrast, RISCSmith automatically derives instruction models from the RISC-V UnifiedDB, ensuring semantic correctness, preparing valid register and memory contexts for more effective testing.

Differential Testing. Differential testing has been widely applied across many system domains. For example, DLFuzz [15] compares the behaviors of different deep learning models to detect anomalies; DIFUZZRTL [17] contrasts CPU RTL execution with a reference simulator to find design flaws; Gandalf [26] checks operator outputs across deep learning libraries to uncover implementation bugs; and CompDiff [24] compares binaries produced by different compilers or optimization levels to reveal issues related to undefined behavior. In contrast, RISCSmith uses low-intrusion instrumentation to collect RTL logs from different CPUs, capturing per-instruction register and memory writes and exception events with minimal overhead. This enables efficient differential comparison while avoiding the inefficiency and potential missed detections of step-by-step pausing, allowing RISCSmith to quickly find the instruction that triggers a divergence and automatically generate a minimized reproducing program.

7 Conclusion

In this paper, we propose RISCSmith, which automatically models rich instructions and performs on-the-fly differential analysis for CPUs. We implemented RISCSmith and evaluated it on six widely used RISC-V CPUs, discovering 18 new bugs and outperforming other state-of-the-art tools in both bug detection and code coverage.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers for their valuable feedback and constructive comments on this paper. This work was supported by the National Key Research and Development Program of China under Grant 2024YFF1401303.

References

[1] Allon Adir, Shady Copty, Shimon Landa, Amir Nahir, Gil Shurek, Avi Ziv, Charles Meissner, and John Schumann. 2011. A unified methodology for pre-silicon verification and post-silicon validation. In *2011 Design, Automation & Test in Europe*. IEEE, 1–6.

[2] Muhammad Nabeel Asghar. 2020. A review of ARM processor architecture history, progress and applications. *Journal of Applied and Emerging Sciences* 10, 2 (2020), pp–171.

[3] RISC-V BOOM. 2025. The Berkeley Out-of-Order RISCV Processor. https://github.com/riscv-boom/riscv-boom. Accessed at October 29, 2025.

[4] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 529–534.

[5] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2023. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 1–12. doi:10.1109/HOST55118.2023.10133714

[6] Jamison D Collins, Dean M Tullsen, Hong Wang, and John Paul Shen. 2001. Dynamic speculative precomputation. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 306–317.

[7] Enfang Cui, Tianzheng Li, and Qian Wei. 2023. Risc-v instruction set architecture extensions: A survey. *IEEE Access* 11 (2023), 24696–24711.

[8] Freddy Gabbay and Avi Mendelson. 1998. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems (TOCS)* 16, 3 (1998), 234–270.

[9] Rocket Chip Generator. 2025. CHIPS Alliance. https://github.com/chipsalliance/rocket-chip. Accessed at October 29, 2025.

[10] GRCOV. 2025. Rust tool to collect and aggregate code coverage data for multiple source files. https://github.com/mozilla/grcov. Accessed at October 29, 2025.

[11] OpenHW Group. 2025. CVA6 RISC-V CPU. https://github.com/openhwgroup/cva6/. Accessed at October 29, 2025.

[12] OpenXiangShan Group. 2025. Open-source high-performance RISC-V processor. https://github.com/OpenXiangShan/XiangShan. Accessed at October 29, 2025.

[13] SpinalHDL Group. 2025. A FPGA friendly 32 bit RISC-V CPU implementation. https://github.com/SpinalHDL/VexRiscv. Accessed at October 29, 2025.

[14] YosysHQ Group. 2025. PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32. Accessed at October 29, 2025.

[15] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

[16] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-level abstraction (ila) a uniform specification for system-on-chip (soc) verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24, 1 (2018), 1–24.

[17] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1286–1303.

[18] Intel. 1994. *Intel Annual Report*. https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html.

[19] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: speculative load hazards boost rowhammer and cache attacks. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 621–637.

[20] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 3219–3236.

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. doi:10.1109/SP.2019.00002

[22] Christos Kyrkou. 2009. Stream processors and GPUs: Architectures for high performance computing. *Survey on Stream Processor and Graphics Processing Units* (2009).

[23] Huimin Li, Nele Mentens, and Stjepan Picek. 2023. Maximizing the potential of custom RISC-V vector extensions for speeding up SHA-3 Hash functions. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[24] Shaohua Li and Zhendong Su. 2023. Finding unstable code via compiler-driven differential testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 238–251.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (May 2020), 46–56. doi:10.1145/3357033

[26] Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. 2023. Generation-based differential fuzzing for deep learning libraries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–28.

[27] Guillermo A Lopez, Michela Taufer, and Patricia J Teller. 2007. Evaluation of IEEE 754 floating-point arithmetic compliance across a wide range of heterogeneous computers. In *Proceedings of the 2007 conference on Diversity in computing*. 1–4.

[28] Karthik Parvathinathan, Maya Dewhurst, Victor Hayashi, and Joshua M Pearce. 2025. Open-Source RISC-V Extension Design: Adding Custom Vector Instructions for DSP Workloads. (2025).

[29] Cosmin-Andrei Popovici, Andrei Stan, Nicolae-Alexandru Botezatu, and Vasile-Ion Manta. 2025. RiscADA: RISC-V Extension for Optimized Control of External D/A and A/D Converters. *Electronics* 14, 15 (2025), 3152.

[30] Spike risc-v isa simulator. 2025. RISC-V Software. https://github.com/riscv-software-src/riscv-isa-sim. Accessed at October 29, 2025.

[31] RISCV-DV. 2025. Random instruction generator for RISC-V processor verification. https://github.com/chipsalliance/riscv-dv. Accessed at October 29, 2025.

[32] Muhammad Sabih, Abrarul Karim, Jakob Wittmann, Frank Hannig, and Jürgen Teich. 2024. Hardware/software co-design of RISC-V extensions for accelerating sparse DNNs on FPGAs. In *2024 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 01–09.

[33] Karthikeyan Sankaralingam, Jaikrishnan Menon, and Emily Blem. 2013. *A detailed analysis of contemporary arm and x86 architectures*. Technical Report.

[34] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade:{CPU} fuzzing via intricate program generation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5341–5358.

[35] SonalPinto. 2025. Kronos is a 3-stage in-order RISC-V core towards FPGA implementations. https://github.com/SonalPinto/kronos. Accessed at October 29, 2025.

[36] Caroline Trippel, Yatin A Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *ACM SIGPLAN Notices* 52, 4 (2017), 119–133.

[37] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A framework for software diversification with {ISA} heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 427–442.

[38] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. {MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1307–1324.

[39] Gen Zhang, Pengfei Wang, Tai Yue, Danjun Liu, Yubei Guo, and Kai Lu. 2024. INSTILLER: Toward efficient and realistic RTL fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 7 (2024), 2177–2190.

[40] Wendi Zhang, Yonghui Zhang, and Kun Zhao. 2021. Design and verification of three-stage pipeline cpu based on risc-v architecture. In *2021 5th Asian Conference on Artificial Intelligence Technology (ACAIT)*. IEEE, 697–703.